DRL-Epanet: Deep reinforcement learning for optimal control at scale in Water Distribution Systems

Belfadil Anas
Techlab Repsol
Universitat Politècnica de Catalunya
Barcelona
anas.belfadil@upc.edu

David Modesto
Barcelona supercomputing center
Universitat Politècnica de Catalunya
Barcelona
david.modesto@bsc.es

Jose Antonio Martin H.
Repsol Techlab
Madrid
ja.martin.h@repsol.com

Abstract

Deep Reinforcement Learning has undergone a revolution in recent years, enabling researchers to tackle a variety of previously inaccessible sequential decision problems. Nevertheless, this method is not widely employed in Water Distribution Systems. In this paper, we demonstrate that DRL can be coupled with the popular hydraulic simulator Epanet and that DRL-Epanet can be applied to a variety of difficult WDS problems. As an example, we use it for pressure control in WDS. We show that DRL-Epanet is scalable to massive action spaces and demonstrate its effectiveness on a problem with more than one million possible actions at each time step. We also demonstrate that it can deal with uncertainties such as stochastic demands, contamination, and other risks; for instance, we address the problem of pressure control in the presence of random pipe breaks. We show that the BDQ algorithm is capable of learning in this context, and we enhance it with an algorithmic modification, BDQF (BDQ with Fixed actions), that achieves better rewards, especially when non-fixed actions are sparse. Finally, we argue that DRL-Epanet can be used for real-time control in smart WDS, which is an advantage over existing methods.

1 Introduction

Water is a limited resource with an increasing number of users. In actuality, the global population has expanded by almost 1.5 billion people in the last 20 years, increasing the demand for clean water. Furthermore, over-exploitation of water resources has been exacerbated by urbanization, climate change, and drought. As a result, municipalities, water utility firms, and societies, in general, must embrace more sustainable water management techniques.

Complex and expanding water networks make it difficult to achieve satisfactory, cost-effective operations. Consequently, researchers have developed novel optimization techniques. Savić et al. (2018) examine deterministic and stochastic (heuristic) optimization in their survey of WDS optimization. The principal deterministic approaches presented are:

- Linear Programming (LP): can find optimal solutions but only works for a continuous problem with a linear objective function subject to linear constraints.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

- Dynamic Programming (DP): suitable for multistage optimization problems, mostly used for pump scheduling. However, Savić et al. (2018) note that this methodology suffers from the so-called 'curse of dimensionality', which limits to some extent its application to large WDS.

- Nonlinear Programming (NLP): similar to LP, NLP works with continuous variables, it is limited in the number of variables and constraints it can handle and thus can only manage WDS of limited size.

Due to the nonlinearity and discreteness of many WDS problems, Savić et al. (2018) note that researchers have moved away from deterministic methods and toward heuristic optimization techniques, which are typically coupled with hydraulic solvers such as Epanet. Principal advantage is that they have solved challenging problems that no problem-specific deterministic algorithm can currently solve efficiently. WDS optimization utilizes a variety of metaheuristics, including genetic algorithms and their variants, simulated annealing, particle swarm optimization, and so on.

Deterministic and stochastic optimization methods face additional challenges in the context of real-time control (RTC), where the optimal set of actions must be determined based on continuous measurements collected in real-time. In this case, a trade-off between method efficiency and precision must be struck, resulting in simplified hydraulic models and/or a very limited computing budget for the optimization procedure, which has an impact on the solution's quality.

Deep Reinforcement Learning (DRL) has revolutionized sequential decision-making in recent years. It has achieved ground-breaking results in several fields, including superhuman performance in Chess and Go, protein folding prediction, control of traffic lights, autonomous driving, and robotic control... DRL has numerous benefits that can be exploited for optimal control in WDS: Scaling to high-dimensional problems, dealing with stochasticity, applying to RTC problems since only inference is required after training, and continuous learning after deployment.

In this paper, we will present a technique for optimizing pressure control in WDS using Deep Reinforcement Learning and the widely used hydraulic software Epanet. We will demonstrate its efficacy in real-world situations involving large action spaces and random pipe breaks. We will argue that this framework is sufficiently general to address a wide variety of sequential decision optimization problems in WDS and can be used for real-time control in intelligent WDS.

2 Related works

EPANET: In Araujo et al. (2006) Epanet was used alongside a genetic algorithm (GA) for optimizing the number of valves and their locations for pressure control in the WDS. In Bonthuys et al. (2020) authors developed an optimization procedure for energy recovery and reduction of leakage utilizing GA with the hydraulic modeling performed in Epanet. Ant colony optimization was used in López-Ibáñez et al. (2008) in conjunction with Epanet for optimal control of pumps in WDS, warnings issued by Epanet for the inefficient operation of pumps were used in the constraint handling procedure. However, the recent survey on control optimization in WDS Mala-Jetmarova et al. (2017) concludes that even with parallel programming techniques and more efficient deterministic optimization methods, WDS simulations may still be computationally prohibitive for Real-Time Control (RTC), especially as the fidelity of the model and the number of decision variables increase.

DRL: Mullapudi et al. (2020) used reinforcement learning for real-time control of stormwater systems, they used the Deep Q-Network algorithm and limited the action space to only 27 possible actions. In water distribution systems, Lee and Labadie (2007) used reinforcement learning for stochastic optimization of multi-reservoir systems, Hajgató et al. (2020) Used DRL for real-time optimization of pumps in WDS, they found that their agent is capable of performing as well as the best conventional techniques while being $2\times$ faster. They also noted the advantage of the DRL for RTC in comparison to previous methods. Mosetlhe et al. (2020) Authors used DRL with a quadratic approximation of WDS hydraulics to solve pressure control using PRVs, their emphasis was on the "model-free" nature of their approach and their treatment of the DRL method used was minimal. To the best of our knowledge, these papers and others that applied DRL for WDS were restricted to small action spaces in order of dozens of possible actions at max.

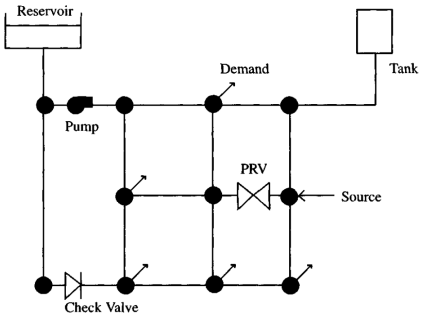


Figure 1: Node-link representation of a WDS

3 Contributions

In order to achieve optimal control in operational WDS, we introduce a heuristic approach that integrates the Epanet solver with Deep Reinforcement Learning. Hajgató et al. (2020) proved that this approach improved pump performance in real time. Here, we use it for pressure control and highlight its generality for other WDS optimal control problems for large action spaces and in a stochastic context. In order to represent the Q-value function, we employ a Reinforcement Learning algorithm based on a branching neural network architecture (BDQ), which is scalable to large action spaces and can be used for discrete action values such as open/close valves and on/off pumps, as well as continuous actions and mixed action spaces.

In nine actual WDS, we demonstrate the effectiveness of this method for pressure control under normal conditions, i.e. no pipe bursts. We demonstrate that the average pressure can be reduced by up to 26%. In addition, we demonstrate the superiority of BDQ over DQN on a high-dimensional action space containing one million actions.

Then, we modify the BDQ algorithm to accommodate random pipe bursts in WDS. We present BDQF, which is superior to BDQ in this scenario, particularly when allowable actions at each step are scarce. To the best of our knowledge, our method for pressure control in the presence of pipe bursts is the first.

We utilize our own optimized OpenAI Gym Brockman et al. (2016) interface of the Epanet solver in Python, as well as our own implementation of the BDQ algorithm, which we contributed to Tianshou Weng et al. (2021), an open source library licensed under the MIT license for reinforcement learning (look for Branching DQN).

4 Framing the WDS pressure control as an RL problem

Water distribution networks can be modeled as a collection of *links* connecting *nodes*. Water flows along links and enters or leaves the system at nodes. All the actual physical components of a distribution system can be represented in terms of these constructs. One particular scheme for accomplishing this is shown in 1. In it, links consist of pipes, pumps, or control valves. Pipes convey water from one point to another, pumps raise the hydraulic head of water, and control valves maintain specific pressure or flow conditions. Nodes consist of pipe junctions, reservoirs, and tanks. Junctions are demand nodes where links connect and where water consumption occurs. Reservoir nodes represent fixed-head boundaries, such as lakes, groundwater aquifers, treatment plant clear wells, or connections to parts of a system not being modeled. Tanks are storage facilities, the volume, and water level of which can change over an extended period of system operation.

Our objective is to train a reinforcement learning agent to control valves in the WDS to minimize the overall pressure in the network under the constraints of minimum and maximum pressure, and without emptying or overfilling the tanks.

We take a time-step of one hour, which is the usual time step for hydraulic simulations in WDS. We consider a full-length episode to be 24 hours, which corresponds to the usual cycle of WDS's operations. At each time step the agent receives the pressures at demand nodes, the tank levels, and

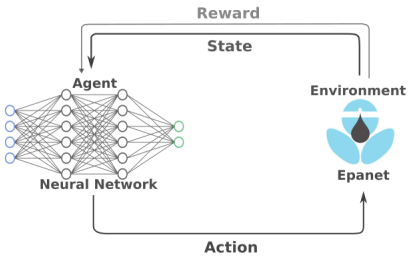


Figure 2: WDS reinforcement learning setup

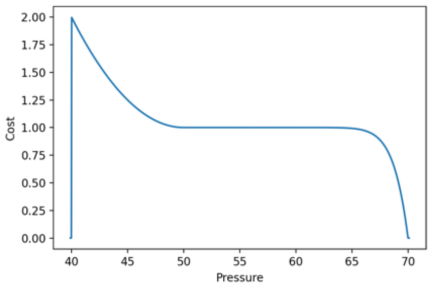


Figure 3: Cost function with smooth transitions.

the time, this is the state s of the environment. It should then decide the opening values of the valves, these are the actions a. A simulation is then performed via Epanet to calculate the new pressures at the demand nodes and the new tank levels: the environment transitions to the new state s'. The reward r is then calculated and returned to the agent. The quadruple $e = (s, a, r, s')$ is called an experience 2.

The rewards

Rewards depend on two terms:

Pressures: Our goal is to insure service for the highest number of clients while maintaining operational constraints. Hence, we use a cost function c_p for counting the nodes that are served with pressures inside the required range. c_p is equal to a number between one and two for nodes with pressures between p_{min} and p_{max} and to zero otherwise, it is biased towards smaller pressures to incentivize pressure reduction, see Figure 3. These values are then summed and normalized over the number of nodes.

Tanks: Over-topping or emptying the tanks are considered to be episode-ending incidents.
The majority of water-distribution networks operate on a 24 h cycle in which the storage tanks are refilled overnight when the charge for electricity is low and then drawn down during the daytime hours when demands are high. This not only reduces the operating costs but also ensures a turnover of the water in storage, thereby avoiding stagnation. It is of most importance to avoid over-topping or emptying the tanks, these events are considered to be episode-ending incidents.

Therefore the reward at time-step t is formalized as:

$$r_t = \sum_{i=1}^{N} \frac{c_p(p_i)}{n}$$

Where p_i is the pressure at demand node i, N is the number of nodes in the WDS.

The goal of the RL agent is to maximize the cumulative discounted reward over the episode:

$$G = \sum_{t=0}^{t=t_f} \gamma^t r_t$$

Where t_f is equal to 23 for a non interrupted episode, otherwise it is the time when the incident of over-topping or emptying any of the tanks occurs.

5 DRL algorithms used

DQN and BDQ

DQN is a classical value-based Deep Reinforcement learning algorithm, it was introduced by Deepmind in 2015 kick-starting the modern revolution in RL, and has been used since then in

numerous publications. However, DQN suffers from the curse of dimensionality, as the network representing the Q-values scales exponentially w.r.t. the number of dimensions in the action space. BDQ Tavakoli et al. (2017) was introduced to solve this problem by adopting a branching architecture that represents each dimension in the action space as a branch, and has a common network "trunk" that coordinates between the branches, see Annex A and B for a detailed presentation of the DQN and BDQ algorithms.

Adapting BDQ for the pipe failure scenario

If a pipe were to burst, some PRVs would have to be turned off for the duration of the episode. No longer can the agent freely manipulate the valves that connect the affected sector. Consistent with previous research Vinyals et al. (2017) Berner et al. (2019) Ye et al. (2019), we consider the entire action space, which includes all possible actions, and employ action masking to consider only valid actions at a given state. While all of these papers account for action masking in policy gradient algorithms, we were unable to locate any that did so for value-based algorithms. However, we found it discussed in a blog post Zouitine (2021), and Tianshou's DQN implementation included it. In both cases, the idea was to assign low or negative Q-values to masked actions so that the policy never selects them $\pi(a|s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$. In this paper we take a different approach by leveraging the branching architecture of BDQ in order to avoid attributing rewards to fixed actions.

Fixing some actions still produces a valid policy For a given pipe failure, let V_f represent the subset of valves that must be closed, A_f represent the elements of V_f that are in the action space, and Anf represent the remainder of the action space. Then, the agent's actions can be represented as: $a = (a_{nf}, a_f)$ with $a_{nf} \in A_{nf}, a_f \in A_f$. The crucial point here is that when action a is performed, only the a_nf portion has effects on the environment (since a_f are fixed and their corresponding valves will remain closed even if the policy wants to assign them different values). Consequently, $r(s,a) = r(s, a_nf)$, and $Q(s,a) = Q(s, a_nf)$ follows, indicating that, learning $Q(s,a)$ is equivalent to learning $Q(s, a_nf)$; therefore, any Q-learning algorithm can generate a valid policy without modification. However, learning $Q(s,a)$ is less efficient and contains unnecessary duplicates, as $Q(s,(a_nf, a_f)) = Q(s,(a_nf, a'_f))$ for any a_f, a'_f. Moreover, in the case of function approximation, this inefficiency will manifest as a learning noise corresponding to the assignment of rewards to ineffectual actions.

Leveraging the benefit of separating the action dimensions in BDQ can mitigate the aforementioned issues. We propose "freezing" action dimensions with fixed values by preventing the back-propagation of the learning signal in action-fixed branches. This will allow the correct Q values to be learned without artificially assigning them low or negative values. We modify the temporal-difference target 9 and the loss function 8 of BDQ as follows:

$$y = r + \gamma \, \frac{1}{N_{nf}} \sum_{d_{nf}} Q'_d(s, \underset{a' \in \mathcal{A}_{d_{nf}}}{\operatorname{argmax}} Q_\theta(s', a')) \tag{1}$$

$$L = E_{(s,a,r,s') \sim \mathcal{D}} \big[\sum_{d_{nf}} \big(y_{d_{nf}} - Q_{d_{nf}}(s, a_{d_{nf}}) \big)^2 \big] \tag{2}$$

Where N_{nf} and d_{nf} are respectively the number and the dimensions of the non-fixed part of the action space. The detailed algorithm can be found in Annex 1

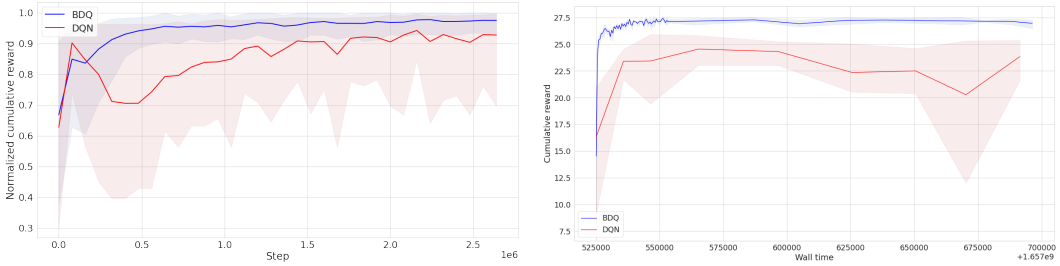
6 Results

Optimal pressure control in normal conditions

We apply the DQN and BDQ algorithms to a set of nine real WDS, of which eight are from Barcelona and one is from Marbella. We operate under normal conditions, with no pipe-bursting incidents. These networks contain either two or four control valves, each of which can be set to any value between 10 and 50 PCA. The action space is discretized by dividing the interval from 10 to 50 p.m. into eight equal bins.

WDS Name	Nodes	Edges	Valves	$\|A\|$	Before	After	Improvement
Maternitat	1902	1964	2	16	59.9	44	26%
Placa Maragall	2370	2463	2	16	55.8	**46.7**	**16%**
Gran de Gracia	2756	2869	2	16	52.1	46.8	10%
Gracia Nova	3153	3274	4	256	55.1	**49.2**	**11%**
Placa Barraquer	1161	1194	2	16	57.6	**49.6**	**14%**
La Prosperitat	1841	1900	2	16	48.9	**47.3**	**3%**
Tuset	1328	1373	2	16	58	47	19%
Marbella	2418	2552	4	256	34.4	**30.1**	**12%**
Virrei Amat	2275	2364	2	16	51.2	**47.9**	**7%**

Table 1: Mean pressures before and after optimization using Deep RL. In **bold** are the cases where BDQ achieved better rewards than DQN.



(a) Comparison of BDQ and DQN on the 9 WDS in table 1 with bootstrapped 95% CI

(b) BDQ vs DQN on the Gracia Nova WDS with 32 bins for each of the 4 valves

Figure 4: BDQ vs DQN, 8 bins and 32 bins

In every WDS, the agents were able to find better solutions than the reference actually used by operators, achieving an improvement of up to 26% in mean pressure, with an average of 13% improvement across all WDS; see figure 5 for a breakdown of the improvement in pressure for each WDS.

The table 1 includes a description of each network as well as the average pressures before and after optimization with Deep RL. Textbfbold indicates instances in which BDQ achieved greater rewards than DQN. In all instances, the best cumulative reward achieved by both algorithms was nearly identical, differing by no more than 0.5%; this is expected given that the action space for these networks is relatively small, with a maximum of $8^4 = 256$ possible actions.

In 4a, we represent a summary of the learning curves of BDQ and DQN across all nine WDS. Each of the 25 randomly seeded runs that were executed required 48 hours to complete. To summarize the plots, we normalized the rewards and calculated bootstrapped confidence intervals in accordance with Agarwal et al. (2021). Even though the best-obtained rewards at the test time are very similar, the plot reveals that BDQ learns more quickly and is more stable than DQN.

When 32 bins are considered in each action dimension for four valves, the resulting action space size is $32^4 = 1\ 048\ 576$. In this scenario, DQN performs significantly worse than BDQ 4b, and if we increase the number of bins to 50, for instance, the 16 Gb Nvidia Tesla v100 GPU that we use could no longer fit the DQN network in its memory. Due to the manner in which DQN networks represent the Q values, with one neuron in the output layer for each possible action, it scales exponentially with the number of dimensions of action space. In contrast, BDQ employs a branching architecture that permits it to scale linearly with the number of action space dimensions. This results in a $7\times$ improvement in performance for BDQ over DQN in the case of 32 bins, where BDQ can execute 7M steps while DQN can only execute 1M with the same computation budget. As we add more control elements to the WDS, the performance gap will grow exponentially, meaning that only BDQ can be used for more complex cases.

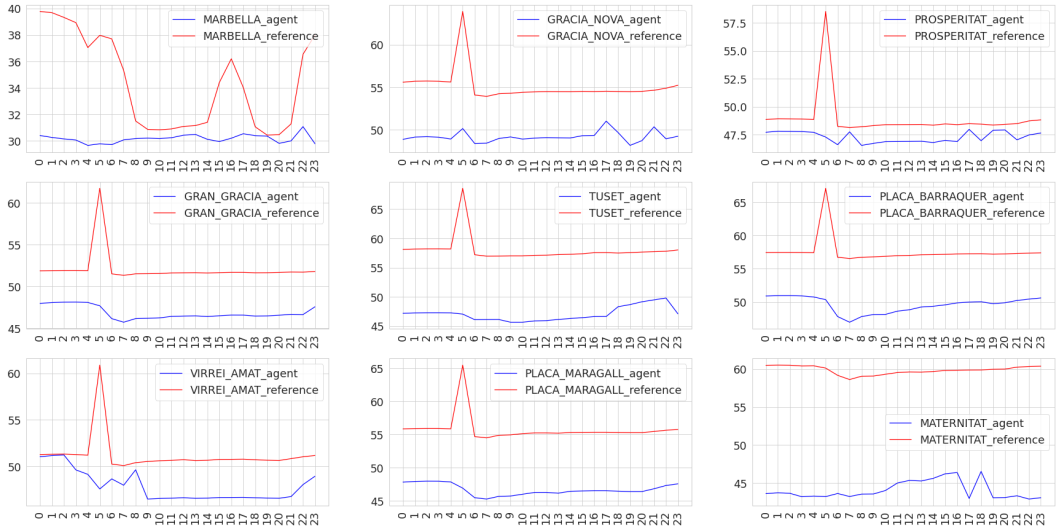
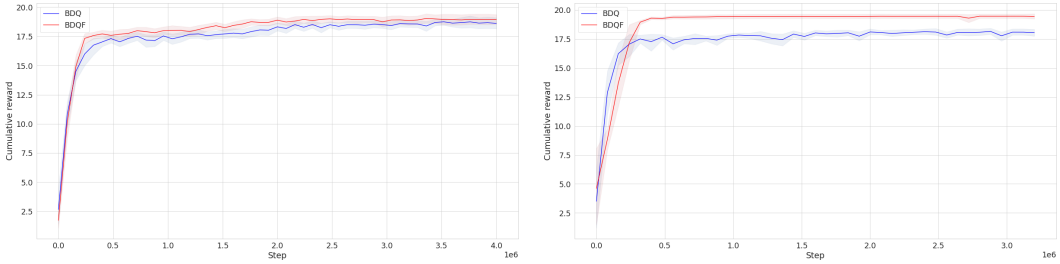


Figure 5: Mean Pressures comparison between the reference solution and the best RL agents solutions from table 1

Optimal pressure control in the presence of random pipe bursts incidents

We consider the scenario of random leaks in the network. The WDS in this case is that of the municipality of Castelldefels; it has 2565 nodes, 2936 links, and 18 open/close control valves that the agent can manipulate to stabilize the pressure. The WDS is divided into ten sectors, each of which can be isolated if a pipe bursts in one of them. Some control valves must remain closed when isolating a sector and cannot be changed by the agent. We can use BDQ without any modifications with some drawbacks, as we mentioned earlier in 5, or we can use our modified version that takes into account fixed actions BDQF. To compare, we train the two agents for 20 randomly seeded runs of 48 hours each. During training, each episode is started by a random pipe burst that corresponds to isolating one of the ten sectors. The agents are evaluated for each of the ten possible pipe burst sectors, and the mean reward is calculated over these evaluations.

In this case, there is no reference solution against which the performance of the agents can be compared. However, we can observe a clear learning tendency for both algorithms, as both BDQ and BDQF were successful in learning good policies that prevent the premature termination of episodes due to emptying/overfilling the tanks and achieve high rewards. BDQF's best runs yielded a 10 sector average reward of 20,2 while BDQ's best yielded 19,3. To evaluate the performance of both algorithms, we plot the bootstrapped 95% CI of the 10 sectors' mean reward over 20 iterations in 6a; we observe that BDQF outperforms BDQ. This is more evident in 6b, where we modified the action space to have sparse free actions that are indeed controlled by the agent.



(a) BDQ vs BDQF for pressure control in presence of random leaks

(b) BDQ vs BDQF for pressure control in presence of random leaks with sparse free actions

Figure 6: BDQ vs BDQF

7 Conclusion and future work

We presented DRL-Epanet a method for optimal pressure control in water distribution systems (WDS) based on Deep Reinforcement Learning and Epanet. We demonstrated the effectiveness of this approach on ten real-world WDS, in all of which the A.I. solutions were superior to the reference solutions used by the operators by up to 26% of improvement in the mean pressure. We demonstrated the advantage of using BDQ over DQN for large action spaces. We then showed that BDQ can be used for optimal pressure control in presence of random pipe bursts, and we made an improvement BDQF over BDQ that mitigated the problem of learning noise introduced by considering gradients of actions that have no effect on the environment.

DRL-Epanet framework can be used to tackle a wide range of sequential decision optimization problems in WDS, such as pressure control as we demonstrated in this paper, pump optimization, energy, water quality, etc... or any combination of the aforementioned problems. We believe that any WDS sequential decision problem that can be simulated using Epanet, can in principle be solved in this framework. Moreover, DRL-Epanet provides a Real-Time Control solution, since once it is trained, the agent is capable to react in real-time to changes in the network with only inference calculation needed, usually in the order of milliseconds, i.e. without recalculating the solution which can be a limiting factor for other methods. We showed in this paper that DRL-Epanet can deal with the stochasticity of the WDS environment in the form of random pipe bursts. Likewise, the DRL-Epanet agent can be trained with more stochastic scenarios to deal with demand uncertainty, contamination, and other component failures scenarios...

In the future, we would like to use model-based DRL to have better sample efficiency. In fact, previous work has shown that Artificial Neural Networks can be used as a substitute to Epanet simulation for WDS with a high degree of accuracy Rao and Alvarruiz (2007), which indicates that model-based DRL can be expected to work quite well for WDS. Another direction for research is using Graph Neural Networks which are the most natural representation for a WDS and can constitute a strong induction bias for DRL that might help in learning more efficiently and lead to better solutions, Hajgató et al. (2021) have shown that such approach works for supervised learning for reconstructing nodal pressure in WDS. We would like also to be able to quantify the uncertainty of the agent actions in production, so we can differ control to the operator in uncharted territory, where the agent is uncertain.

References

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., and Bellemare, M. G. (2021). Deep reinforcement learning at the edge of the statistical precipice. *CoRR*, abs/2108.13264.

Araujo, L., Ramos, H., and Coelho, S. (2006). Pressure control for leakage minimisation in water distribution systems management. *Water resources management*, 20(1):133–149.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680.

Bonthuys, G. J., Van Dijk, M., and Cavazzini, G. (2020). Energy recovery and leakage-reduction optimization of water distribution systems using hydro turbines. *Journal of Water Resources Planning and Management*, 146(5):04020026.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Hajgató, G., Gyires-Tóth, B., and Paál, G. (2021). Reconstructing nodal pressures in water distribution systems with graph neural networks. *CoRR*, abs/2104.13619.

Hajgató, G., Paál, G., and Gyires-Tóth, B. (2020). Deep reinforcement learning for real-time optimization of pumps in water distribution systems. *Journal of Water Resources Planning and Management*, 146(11):04020079.

Lee, J.-H. and Labadie, J. W. (2007). Stochastic optimization of multireservoir systems via reinforcement learning. *Water Resources Research*, 43(11).

López-Ibáñez, M., Prasad, T. D., and Paechter, B. (2008). Ant colony optimization for optimal control of pumps in water distribution networks. *Journal of water resources planning and management*, 134(4):337–346.

Mala-Jetmarova, H., Sultanova, N., and Savic, D. (2017). Lost in optimisation of water distribution systems? a literature review of system operation. *Environmental Modelling Software*, 93:209–254.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Mosetlhe, T. C., Hamam, Y., Du, S., Monacelli, E., and Yusuff, A. A. (2020). Towards model-free pressure control in water distribution networks. *Water*, 12(10).

Mullapudi, A., Lewis, M. J., Gruden, C. L., and Kerkez, B. (2020). Deep reinforcement learning for the real time control of stormwater systems. *Advances in Water Resources*, 140:103600.

Rao, Z. and Alvarruiz, F. (2007). Use of an artificial neural network to capture the domain knowledge of a conventional hydraulic simulation model. *Journal of Hydroinformatics*, 9(1):15–24.

Savić, D., Mala-Jetmarova, H., et al. (2018). History of optimization in water distribution system analysis:(009). In *WDSA/CCWI Joint Conference Proceedings*, volume 1.

Tavakoli, A., Pardo, F., and Kormushev, P. (2017). Action branching architectures for deep reinforcement learning. *CoRR*, abs/1711.08946.

Tsitsiklis, J. and Van Roy, B. (1996). Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J. P., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. (2017). Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR.

Weng, J., Chen, H., Yan, D., You, K., Duburcq, A., Zhang, M., Su, H., and Zhu, J. (2021). Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*.

Ye, D., Liu, Z., Sun, M., Shi, B., Zhao, P., Wu, H., Yu, H., Yang, S., Wu, X., Guo, Q., Chen, Q., Yin, Y., Zhang, H., Shi, T., Wang, L., Fu, Q., Yang, W., and Huang, L. (2019). Mastering complex control in MOBA games with deep reinforcement learning. *CoRR*, abs/1912.09729.

Zouitine, A. (2021). Masking in deep reinforcement learning.

A DQN: Deep Q-Network algorithm

One of the most significant advancements in reinforcement learning was the development of the deep Q network algorithm, Mnih et al. Mnih et al. (2015). It adds the power of deep neural networks to Q-learning. In fact, Q-Learning has been proven to converge toward the optimal solution in a tabular case or using linear function approximation. However, large and continuous state and action spaces need non-linear function approximators such as a neural network, and it is known that Q-learning is unstable or even diverges when using such an approximator Tsitsiklis and Van Roy (1996). With the advances in training deep neural networks, Mnih et al. in their paper *Human-level control through deep reinforcement learning* Mnih et al. (2015) addressed this problem and reconciled Q-learning with deep neural networks. Thus, giving birth to DQN and igniting research in reinforcement learning.

A.1 DQN

DQN is the first scalable reinforcement learning algorithm that combines successfully Q-learning with deep neural networks. To overcome stability issues, DQN adopts the following new techniques that turned out to be essential for the balance of the algorithm.

A.1.1 Replay memory

To use more IID data during SGD iterations, DQN introduced a replay memory (also called experienced replay) to collect and store the experience in a large buffer. This buffer ideally contains as many experiences as possible. When doing SGD, a random mini-batch will be gathered from the experienced replay and used in the optimization procedure. Since the replay memory buffer holds varied experiences, the mini-batch that's sampled from it will be diverse enough to provide independent samples. Another very important feature behind the use of an experience replay is that it enables the reusability of the data as the experiences will be sampled multiple times. This greatly increases the data efficiency of the algorithm compared to on-policy algorithms.

A.1.2 Target network

The moving target problem is due to continuously updating the network during training, which also modifies the target values. Nevertheless, the neural network has to update itself in order to provide the best possible state-action values. The solution that's employed in DQNs is to use two neural networks. One is called the online network, which is constantly updated, while the other is called the target network, which is updated only every N iteration (with N usually being between 1,000 and 10,000). The online network is used to interact with the environment while the target network is used to predict the target values. In this way, for N iterations, the target values that are produced by the target network remain fixed, preventing the propagation of instabilities and decreasing the risk of divergence. A potential disadvantage is that the target network is an old version of the online network. Nonetheless, in practice, the advantages greatly outweigh the disadvantages and the stability of the algorithm improves significantly.

A.1.3 The DQN algorithm

With the further employment of the separate Q-target network Q' with weight θ', and the learning update applied to mini-batches drawn uniformly from the experienced buffer. The loss function is expressed as:

$$L(\theta) = E_{(s,a,r,s')\sim\mathcal{D}}\big[\big(r + \max_{a'} Q'_{\theta'}(s',a') - Q_\theta(s,a)\big)^2\big] \tag{3}$$

An improvement over was introduced by Van Hasselt, Guez, and Silver in Van Hasselt et al. (2016), to address the over-optimism in the Q-value estimations by using the current Q-network to select the next greedy action, but evaluating it using the target network. Equation 3), then become:

$$L(\theta) = E_{(s,a,r,s')\sim\mathcal{D}}\big[\big(r + Q'_{\theta'}(s,\ \operatorname*{argmax}_{a'} Q_\theta(s',a')) - Q_\theta(s,a)\big)^2\big] \tag{4}$$

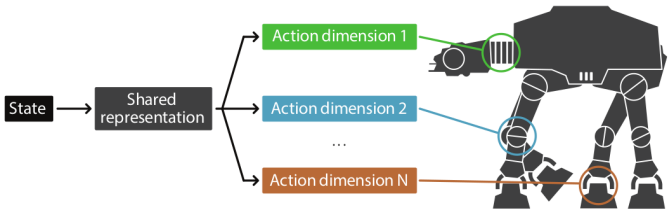


Figure 7: From Tavakoli et al. (2017) A conceptual illustration of the proposed action branching network architecture. The shared network module computes a latent representation of the input state that is then passed forward to the several action branches. Each action branch is responsible for controlling an individual degree of freedom and the concatenation of the selected sub-actions results in a joint-action tuple.

A.1.4 The curse of dimensionality in the DQN algorithm

Despite the success of the DQN algorithm which has enabled the use of reinforcement learning in domains with unprocessed, high-dimensional input. Its application to high-dimensional action spaces continues to suffer from the same issues as in tabular reinforcement learning, that is, the number of actions that need to be explicitly represented grows exponentially with increasing action dimensionality.

This is the case of our WDS environments, where the number of actions grows exponentially with the number of valves. This becomes a serious drawback of the DQN algorithm for a number of valves as small as 20, which corresponds to a number of actions of 1 048 576 (for discrete actions), and therefore needs a NN with 1 048 576 output cells. This renders the DQN algorithm impractical for large numbers of valves or for continuous actions.

B BDQ: Branching dueling Q-Network

In Tavakoli et al. (2017) A. Tavakoli, F. Pardo, and P. Kormushev introduced a novel neural architecture that enables the use of discrete-action algorithms in deep reinforcement learning for domains with high-dimensional discrete or continuous action spaces. The core notion of the proposed architecture is to distribute the representation of the action controllers across individual network branches, meanwhile, maintaining a shared decision module among them to encode a latent representation of the input and help with the coordination of the branches (see Figure 7). The proposed decomposition of the actions enables the linear growth of the total number of network outputs with increasing action dimensionality as opposed to the combinatorial growth in discrete-action algorithms such as DQN.

To demonstrate the capabilities of this new network architecture, the authors used it in implementing a new variant of the dueling DQN algorithm called Branching Dueling Q-network (BDQ). Which is, as it name suggests, an adaptation of the dueling DQN with the branching architecture. They were able to achieve state-of-the-art results on many challenging benchmarks where other discrete action algorithms have failed.

B.1 Action branching architecture

The key insight behind the proposed action branching architecture is that for solving problems in multidimensional action spaces, it is possible to optimize for each action dimension with a degree of independence. However, it is well-known that the naive distribution of the value function or the policy representation across several independent function approximators is subject to numerous challenges which can lead to convergence problems Tavakoli et al. (2017). To address this, the proposed neural architecture distributes the representation of the value function or the policy across several network branches while keeping a shared decision module among them to encode a latent representation of the common input state (see Figure 7). This kind of architecture can also be found in nature. Octopuses, for instance, have complex neural systems where each arm is able to function with a degree of autonomy and even respond to stimuli after being detached from the central control.

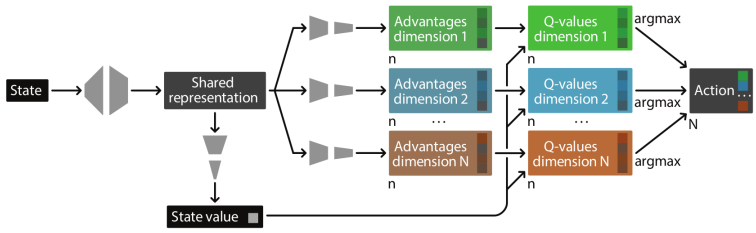


Figure 8: From Tavakoli et al. (2017) action branching network used in the BDQ algorithm.

B.2 BDQ

Besides the improvements on the vanilla DQN algorithm we presented earlier, two more extensions were incorporated into the BDQ agent:

- The dueling architecture
- The prioritized replay buffer

B.2.1 Dueling

For some states, different actions are not relevant to the expected value, and we do not need to learn the effect of each action for such states. So decoupling the action-independent value of state and Q-value may lead to more robust learning. Dueling DQN proposes a new network architecture to achieve this idea (Wang et al. Wang et al. (2016)). More precisely, the Q-value can be split into the state value part and action advantage part as follows:

$$Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a) \tag{5}$$

and dueling DQN separates the representations of these two parts by

$$Q_\pi(s,a;\theta,\theta_V,\theta_A) = V_\pi(s;\theta,\theta_V) + \left(A_\pi(s,a;\theta,\theta_A) - \frac{1}{|\mathcal{A}|}\sum_{a'} A_\pi(s,a';\theta,\theta_A)\right) \tag{6}$$

Where θ_V and θ_A are parameters of two streams of layers, one for the state value $V_\pi(s)$ and the other for the state-action advantage $A_\pi(s,a)$. Subtracting the mean of the advantages from each individual advantage and then summing them with the state value results in improved learning stability when compared to the naive summation of the state value and advantages.

The dueling network architecture has been shown to lead to better policy evaluation in the presence of many similar-valued (or redundant) actions, and thus achieves faster generalization over large action spaces.

B.3 Prioritized experience replay

Instead of sampling uniformly from the replay buffer, PER samples more frequently transition with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error.

B.3.1 Adapting the DQN algorithm

Common state-value estimator As demonstrated in the action branching network of Figure 8, to adapt the dueling architecture into the action branching network, the representation of the (state-dependent) action advantages is distributed on the several action branches, meanwhile, adding a single additional branch for estimating the state-value function. Similar to the dueling architecture, the advantages and the state value are combined, via a special aggregating layer, to produce estimates of the distributed action values. The best performing method is to locally subtract each branch's mean advantage from its sub-action advantages, prior to their summation with the state value.

Formally, for an action dimension $d \in \{1, ..., N\}$ with $|\mathcal{A}_d| = n$ discrete sub-actions, the individual branch's Q-value at state s and sub-action a_d is expressed in terms of the common state value V(s) and the corresponding (state-dependent) sub-action advantage $A_d(s, a_d)$ by:

$$Q_d(s, a_d) = V(s) + \big(A_d(s, a_d) - \frac{1}{n} \sum_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)\big) \tag{7}$$

Temporal-difference target Three different methods were proposed by the paper Tavakoli et al. (2017). The first approach is to calculate a TD target, for each individual action dimension separately. The second is to consider the same target for all the branches, by taking the maximum over the branches. And finally, take the mean over the branches instead of the maximum for a common target for all the branches. They reported the third option to be the best in their experiments:

$$y = r + \gamma \, \frac{1}{N} \sum_d Q'_d(s, \operatorname*{argmax}_{a' \in \mathcal{A}_d} Q_\theta(s', a')) \tag{8}$$

Loss function To avoid the canceling out of the branches errors of different signs, the loss function is defined as the mean squared TD error across the branches:

$$L = E_{(s,a,r,s') \sim \mathcal{D}} \big[\sum_d \big(y_d - Q_d(s, a_d)\big)^2 \big] \tag{9}$$

where $\mathcal{D}$ denotes a (prioritized) experience replay buffer and a denotes the joint-action tuple $(a_1, a_2, ..., a_N)$.

Error for experience prioritization A simple way of aggregating the branches' TD errors is to take the unified prioritization error to be the sum of a transition's absolute, distributed TD errors:

$$e_{\mathcal{D}}(s, a, r, s') = \sum_d |y_d - Q_d(s, a_d)| \tag{10}$$

C Hyper-parameters

C.1 DQN

Hyper-parameter	Value
hidden-sizes	[512, 256]
epsilon-train	0.73
epsilon-test	0.01
epsilon-decay	5×10^-6
buffer-size	10^5
learning-rate	$8e - 5$
gamma	0.99
target-update-freq	500
epoch	1000
step-per-epoch	8×10^4
step-per-collect	24
update-per-step	1/24
batch-size	128
training-environments-number	24
test-environments-number	1

C.2 BDQ and BDQF

Hyper-parameter	Value
common-hidden-sizes	[512, 256]
action-hidden-sizes	128
value-hidden-sizes	128
epsilon-train	0.73
epsilon-test	0.01
epsilon-decay	5×10^-6
buffer-size	10^5
learning-rate	$8e-5$
gamma	0.99
target-update-freq	500
epoch	1000
step-per-epoch	8×10^4
step-per-collect	24
update-per-step	1/24
batch-size	128
training-environments-number	24
test-environments-number	1

D Compute resources

The runs were performed on the CTE-power9 cluster of Barcelona Supercomputing Center, each run was using one Tesla V100 GPU and 40 CPUs to run 24 Epanet environments in parallel as well as the learning algorithm. There were at most 10 running at the same time. The run-time was on the horizon of 48hours.

Algorithm 1: BDQF (BDQ for Fixed actions)

Hyperparameters: replay buffer capacity M, reward discount factor γ, delayed steps C for target network update, ϵ-greedy factor ϵ, number of steps T for testing the model
Inputs: number of episodes N, environment Env takes in the current state-action pair and outputs the reward and next state
Initialize parameters θ of action-value function Q
Initialize target network Q' with parameters $\theta' \leftarrow \theta$
Fill the replay buffer $\mathcal{D}$
For episode = 0,1,2,...,N **do**
Initialize environment with leak, get fixed actions a_f and initial state s_0
  For step t = 0,1,2... **do**
    With probability ϵ select a random action vector a_t, otherwise select $a_t = \left(\underset{a_d \in \mathcal{A}_d}{argmax}\ Q_\theta(s_t, a_d)\right)$
    $r_t, s_{t+1} \leftarrow Env(s_t, a_t)$
    If the episode has ended, set $f_t = 1$. Otherwise set $f_t = 0$
    Store experience $E = (a_t, a_t, r_t, f_t, s_{t+1})$ in $\mathcal{D}$ FIFO
    Sample prioritized minibatch of transitions $E = (s_i, a_i, r_i, f_i, s_{i+1})$ from $\mathcal{D}$
    If $f_i = 0$, set the common target $y_i = r_i + \gamma \frac{1}{N_{nf}} \sum_{d_{nf}} Q'_d(s, \underset{a' \in \mathcal{A}_{d_{nf}}}{\text{argmax}}\ Q_\theta(s', a'))$. Otherwise, set $y_i = r_i$
    Perform a gradient descent step on $\sum_{d_{nf}} \left(y_{i_{d_{nf}}} - Q_{d_{nf}}(s_i, a_{i_{d_{nf}}})\right)^2$
    Synchronize the target Q' every C steps
    Evaluate and save the model every T steps
    If the episode has ended, break the loop
  End for
End for

E NeurIPS questions

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? **[TODO]**yes
   (b) Did you describe the limitations of your work? **[TODO]**NA
   (c) Did you discuss any potential negative social impacts of your work? **[TODO]**NA
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? **[TODO]**yes
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? **[TODO]**yes
   (b) Did you include complete proofs of all theoretical results? **[TODO]**yes
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? **[TODO]**No, the data is proprietary
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? **[TODO]**yes
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? **[TODO]**yes
   (d) Did you include the total amount of computing and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? **[TODO]**yes
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? **[TODO]**yes
   (b) Did you mention the license of the assets? **[TODO]**yes

(c) Did you include any new assets either in the supplemental material or as a URL? **[TODO]**NA

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? **[TODO]**yes

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? **[TODO]**NA

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? **[TODO]**NA

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? **[TODO]**NA

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? **[TODO]**NA